Workshop on Resilient Financial Information Systems
Securing Software Environments

Self-healing Software Systems

Angelos Keromytis
Columbia University
angelos@cs.columbia.edu
Overview

• Create software systems that can "heal" themselves against previously unknown attacks and failures with no (or limited) human intervention

• Systems approach

• Maintain integrity and availability
  • Both are important in a critical infrastructure
  • Performance always desired (otherwise impractical)
Rationale

• Secure-by-design/development is extremely difficult (unattainable?)

• Patch-and-pray is losing the race
  ▶ Symantec: 48 new vulnerabilities/week in 2004
  ▶ Disclosure-to-exploit time: 5.8 days (avg.)
  ▶ Witty worm appeared < 48 hrs after vulnerability announced
  ▶ MyDoom may be the first zero-day worm

• Runtime protection techniques
  ▶ Attack-specific, expensive, ad hoc
  ▶ Trade off integrity, availability and performance

• Sandboxing/isolation
  ▶ Good for confidentiality, not for integrity/availability
Basic approach

- **Automated OODA feedback loop**
  - Sensors that detect instances of specific classes of faults
  - Runtime detection
  - Analysis of fault
  - Remediation
- **Self-testing component**
Self-patching network services

- First proof of concept prototype
- Targets problem of worms/remote attacks using zero-day buffer overflow attacks
  - Attacks "easy" to recognize
  - Generalizable to other types of low-level faults (?)
- Observation: given a program and an exploit, we can "easily" determine whether the bug exists, and what it does
- Intuition: let the attacker (e.g., worm) tell us what/where the bug is
Architectural summary

- Detect suspicious traffic
  - Worm infection vector, remote attack
- Analyze in isolated environment
  - Detect anomaly or recognize class of attack
- Try software fixes
  - Modify source code, recompile, test
- Update production server with patched version
- All the steps occur automatically
Self-patching system architecture
Limitations

• **Source code availability**
  - What about "black box" applications?

• **Generalization (other kinds of faults/attacks)**
  - e.g., application-level DoS

• **Side effects of execution are not caught**

• **Correctness of patched binary**
  - Code with side effects
    - e.g., stdio library functions
  - Human in the loop?

• **Server still gets compromised**
  - Combine with lightweight protection mechanism?
Selective Transactional EMulation

- Second prototype of self-healing concept
- Instruction-level emulator
  - Can be invoked for arbitrary slices of code
  - Switch between emulated and native execution at will
  - Record state and rollback if needed
- Check for faults before instruction is "executed"
- Invoke appropriate remediation strategy
STEM use

• Once we identify vulnerable code, emulate it
• While emulating
  ‣ Record all memory writes
  ‣ Check for operand correctness before emulating instruction
• If failure is detected, rollback memory
  ‣ And abort top function
• Impose transactional semantics to parts of program execution
Application Communities

- Monoculture, black box software
  - Large number of software instances
  - Take advantage of monoculture vs. artificial diversity

- Create a distributed app-pot (akin to a honeypot)
  - Exchange information about newly detected faults
Properties

• Share the burden of monitoring for faults
  ▶ Tradeoff between coverage and system-wide overhead

• Exchange verifiable fault information
  ▶ Attack can be replayed in local environment
Other questions

- What other classes of faults can we catch
  - Application DoS, failures, etc.
- General applicability to COTS
- Correctness
- Other remediation strategies
- Can we learn about new types of attacks
- Program slicing strategies
- Performance implications
- Fairness (for ACs)